VNUHCM-University of Science

CS163 PROJECT REPORT

Topic:

Mini search engine

Group 6 – 20CTT1:

1. Nguyễn Quang Long
2. Trần Minh Nam
3. Văn Hoàng Yến
4. Nguyễn Hoài Nam Phương

Contents

1) *Design Issue*

Our project is building a mini search engine to search for documents that satisfied the users. Therefore, there are many small problems below to handle:

a) *Data loading issue*

We have to load nearly 11500 text files at the beginning of the program. It is a huge amount of files to store directly as a string for every document so we have to handle the storing process efficiently. Moreover, the articles may have special char so we have to remove them before move it to our program storage.

b) *Query issue*

We have more than ten types of queries to handle. The query is the only thing to connect the users with this program so we have to specially handle this program.

c) *History issue*

Every time users searching for something, we need to save that query and display it as a suggestion at the next time they enter that query.

d) *User Interface and inputting issue:*

This is what users see and interact with our program, so we also mark this as important.

e) *Group 100 data files*

We collected 100 data text files from http://www.textfiles.com/ and named every file from Group06_001.txt to Group06_100.txt

f) *Stopword and Synonym*

Our search engine has to carry the stop words and synonyms. The stop words data source is https://www.ranks.nl/stopwords, and synonym data source is https://www.gutenberg.org/files/51155/51155-h/51155-h.htm.

➔ These are problems we have to solve to complete our program

2) *Data structure to solve the problem*

To solve this problem, we first think about how to store the information/the words of each document into the program. Each file has more than 500 words or 3000 characters so storing the information of each file as a slot in an array is not efficient in searching because if we search for a word "ABC", we have to loop through 3000 characters x 11000 files. It costs us a huge comuter resources and time to do that process. Hence, we have to design our search engine with another data structure to store the whole library.

We choose to use trie – a retrieval data structure – because we can merge many word with same prefix together. Therefore, our storing data and searching for keywords will be efficiently optimized.

```
struct node {
    vector<int> title; // Which article has this keyword in title
    vector< pair<int, int> > position; // article and position of its
    vector<string> synonym; // If it's a synonym trie, this vector con-
tains all the synonym of current keyword
    vector<node*> pNext; // '0' - > '9', 'a' -> 'z', '#', '$'
    bool end;
};
```

In each node of a tree, we have:

- A vector of pair "position" to store the article id and word id in the document.

- The “title” vector is for storing the id of the article that the word belong to that article title.
- “pNext” is the pointer to next node. If pNext[id] is not null, the next character in our word is the character that id represent.
- We have a bool to mark this node is the end of a word. This bool value use for searching.
- “synonym” vector to store the synonyms of the current keyword.

3) Algorithm

Our trie have two process, inserting and searching.

a) Inserting a word into a trie:

There are two types of insert: insert a word in document and insert a word in the tile of the document.

The general insertion is starting from the root, we traverse each character of S, and move along the trie with the corresponding character. At the end of the string, marked the bool is true.

Time complexity: $O(length\ of\ S)$

```
void insertTrie(node*& root, string &s, int curArticle) {
    /*
        Insert a string into the trie
        root: Root node of the Trie
        string s: The word to be inserted
        curARticle: The id of the current article that has string s in title
    */
    node *cur = root;
    for (int i = 0; i < s.size(); i++) {
        char c = s[i]; int nxt = _valChar(c);
        if (!cur -> pNext[nxt]) cur -> pNext[nxt] = new node();
        cur = cur -> pNext[nxt];

        if (i == s.size() - 1) cur -> title.push_back(curArticle);
    }
}

void insertTrie(node*& root, string &s, int curArticle, int posInTitle) {
    /*
        Insert a string into the trie
        root: Root node of the Trie
        string s: The word to be inserted
        posInTitle: Position of the string, in the current article
    */
    node *cur = root;
    for (int i = 0; i < s.size(); i++) {
        char c = s[i]; int nxt = _valChar(c);
        if (!cur -> pNext[nxt]) cur -> pNext[nxt] = new node();
        cur = cur -> pNext[nxt];

        if (i == s.size() - 1) cur -
> position.push_back(make_pair(curArticle, posInTitle));
    }
}
```

```
void insertTrie(node*& root, string &s, string &synonym) {
    /*
        Insert a string into the trie
        root: Root node of the Trie
        string s: The word to be inserted
        string synonym: synonym of s
    */
    node *cur = root;
    for (int i = 0; i < s.size(); i++) {
        char c = s[i]; int nxt = _valChar(c);
        if (!cur -> pNext[nxt]) cur -> pNext[nxt] = new node();
        cur = cur -> pNext[nxt];

        if (i == s.size() - 1) cur -> synonym.push_back(synonym);
    }
}
```

The time complexity for searching a string S is :$O(length\ of\ S)$

b) Searching a word in a trie

Start from the root, traverse each character of S. If there is a node, then move along, else we can't find the keyword.

Time complexity: $O(length\ of\ S)$

```
node* searchTrie(node* root, string &s) {
    node *cur = root;
    for (int i = 0; i < s.size(); i++) {
        char c = s[i]; int nxt = _valChar(c);
        if (!cur -> pNext[nxt]) return nullptr;
        cur = cur -> pNext[nxt];

        if (i == s.size() - 1) return cur;
    }
}
```

c) Hashing function:

To convert a char into the id of the trie node as below:

- 0 to 9 is id 0 to 9, respectively

- a to z is id 10 to 36, respectively
- # is id 37
- $ is id 38
- - is id 39

```
int _valChar(char c) {
    int cur = 0;
    if ('0' <= c && c <= '9') return c - '0' + cur;
    cur += 10;
    if ('a' <= c && c <= 'z') return c - 'a' + cur;
    cur += 26;
    if (c == '#') return cur;
    cur++;
    if (c == '$') return cur;
    return 0;
    // If reach this state, then c is lmao
}
```

d) Loading data

We loop through each line of the ___index.txt file, each line in the file is a name of a document. Then we will open each txt file with its name and load data into our trie. We also load synonyms of each word in synonym.txt file.

We have another trie to load, a stop words trie from file stopword.txt.

e) Query preprocessing

From the raw query in the input, we design a fuction to break it into vector of words using string stream and remove all special characters, except the special chars of the query operators.

Example:

User Input	Query After Preprocessing (each word in the vector is separate by '/ ')
ABC	abc

String-stream	string/stream
manchesTEr -UniTED	manchester/-/united
hamburger AND chips	hamburger/AND/chips

We split the operator as a word and do not normalize it.

f) *Special char removal when loading data file*

We get each line in the data file. Then we loop through each line by char, if it is a special char, we skip it. If it is a nomal digit, we add the char into the tmp string. If it is a blank space, we add the tmp string into our trie and reset the tmp string.

To solve the problem of searching many keywords in the query, we design two functions: get intersection and get union. These two functions will also become the core of our query searching.

g) *Get Intersection*

To get the intersection integer value of two vector arrays:

```
vector<int> getIntersection(vector<int>& a, vector<int>& b) {
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    vector<int> res(a.size() + b.size());

    vector<int>::iterator itr = set_intersection(a.begin(), a.end(), b.begin(), b
.end(), res.begin());
    res.resize(itr - res.begin());
    return res;
}
```

h) *Get Union*

To get the union value of two vector arrays

```
vector<int> getUnion(vector<int>& a, vector<int>& b) {
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());
    vector<int> res(a.size() + b.size());

    vector<int>::iterator itr = set_union(a.begin(), a.end(), b.begin(), b.end(),
 res.begin());
    res.resize(itr - res.begin());
    return res;
}
```

To solve our query, we have design the following function:

i) *AND query*

Solving the query AND, by getting the intersection of two vector array.

```
vector<int> AndOperator(vector<int>& res, node* keywordNode) {
    vector<int> tmp;
    if (keywordNode == nullptr)
        return {};
    int n = keywordNode->position.size();
    for (int i = 0; i < n; i++) {
        tmp.push_back(keywordNode->position[i].first);
    }
    res = getIntersection(res, tmp);
    return res;
}
```

j) *Normal query*

This type of query is the query with no operator. We will also use AND operator to solve this normal query

k) *OR query*

Solving query OR, using function getUnion above

```
vector<int> OrOperator(vector<int>& res, node* keywordNode) {
    vector<int> tmp;
    if (keywordNode == nullptr)
        return res;
    int n = keywordNode->position.size();
    for (int i = 0; i < n; i++) {
        tmp.push_back(keywordNode->position[i].first);
    }
    res = getUnion(res, tmp);
    return res;
}
```

l) *Not include query (-)*

We will search the word and get result like a normal word below. Then for every id in the not include vector, we remove it from the answer query

m) *intitle: query*

We get the keyword after this query and get the result vector of that keyword. Then we use function getIntersection of the result vector with this intitle vector.

n) *filetype: query*

Because all data files are txt so it will play as a normal query

o) *Search for a price $*

This query will be considered as a string of USD dollar and combined with the previous result vector using getIntersection function

Our trie node will have a node that present character #

p) *Search hashtag #*

As same as a normal query, we search this query and combined with previous result vector using getIntersection function

Our trie node will have a node that present character $

q) *Search exact match (including wildcard)*

We will split the input query like normal, but keeping the double quote marks. Then, when searching, if we meet a word with the beginning character is the opening double quote, we will jump into the special IF case. We jump out this special case when we reach the word that it ends with ending double quote.

The result of the vector will be combined with the previous result using getIntersection function.

r) *Search for a range of price*

$money1..$money2

I have a function isRangeMoney(string s) to consider if a string in query is a range money query.

Because a range of price have two character $ so whenever it satisfied this condition, I will split it into two numbers. I run for each integer value from money1 to money2 and getUnion for each result vector.

Finally, I use getIntersection function between the previous result vector and the range money result vector.

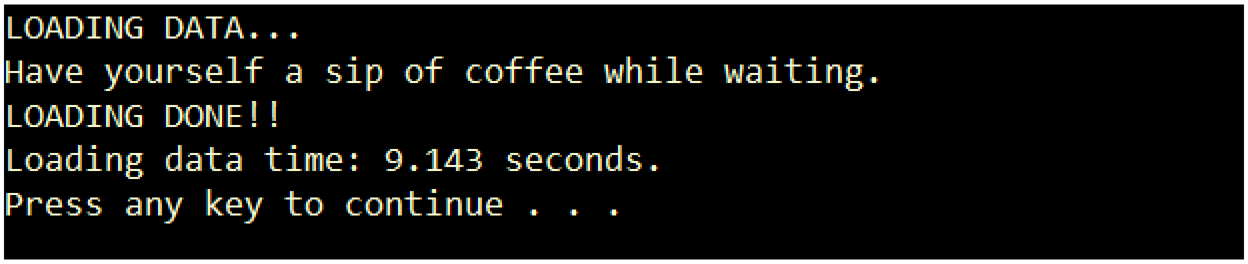
s) *Search with synonyms ~*

~keyword

In this query, we search for the string after character ~ in the synonym trie. We got the vector of string (which is the synonym for keyword) and add it to query vector. We continue the search like normal.

4) ***Running time***

At first, when we open the program, it takes us from about nearly 10 seconds (on Nguyen Quang Long computer) by loading 11500 files.



```
LOADING DATA...
Have yourself a sip of coffee while waiting.
LOADING DONE!!
Loading data time: 9.143 seconds.
Press any key to continue . . .
```

For searching, we get the result immediately if the query is short (below 1ms). In case that the query is so long such as (a AND b AND c OR d $500 “limit time” intitle:E), it will take us . We show you the list of documents that is satisfied with your input query. You will choose which documents you want to see the data inside.

5) ***What to do further to optimize your algorithm***

With loading time, we cannot optimize more. However, we still think about after searching for a query, should we save that result vector. In the future, when user type the same query, we do not have to search again. We just use our saved result vector.

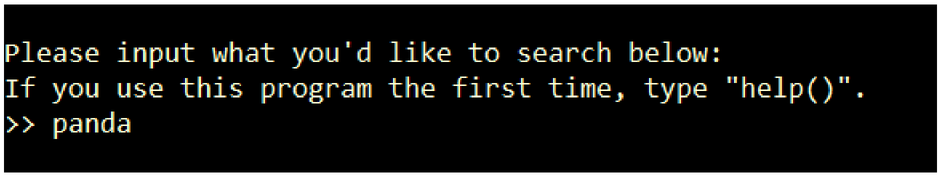
6) ***Is this program efficient in the case of very large text collections?***

Our data has nearly 11500 files, each file has above 2500 characters (500 words). The loading process with this data set takes us only 10 seconds, so I think with bigger dataset, this program can work well for loading

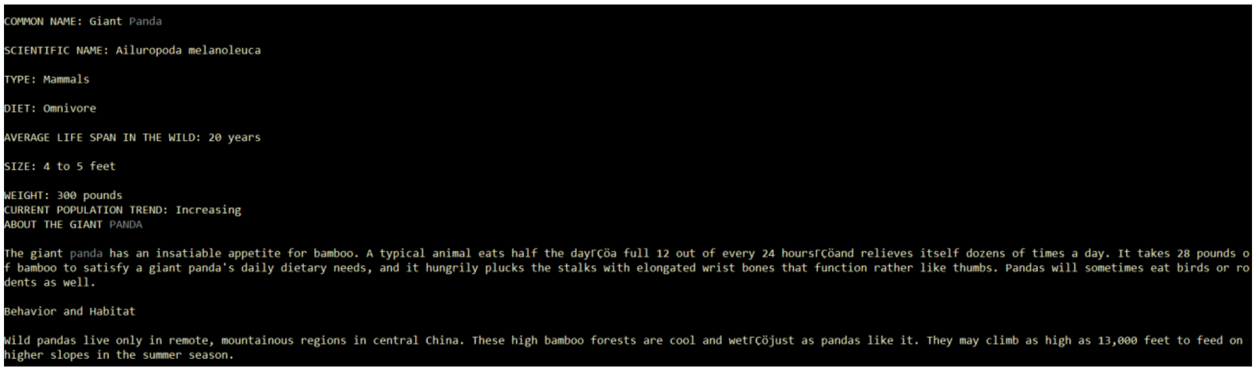
For searching, each word we get the id vector and we merge those vectors together according to the query operator so our program may work well with huge data set.

7) ***Sample query demo***

a) Demo 1: Normal keyword

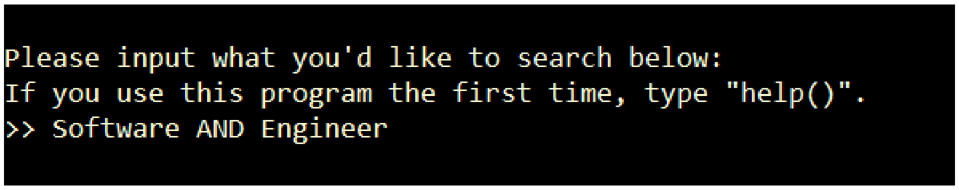


```
Please input what you'd like to search below:
If you use this program the first time, type "help()".
>> panda
```

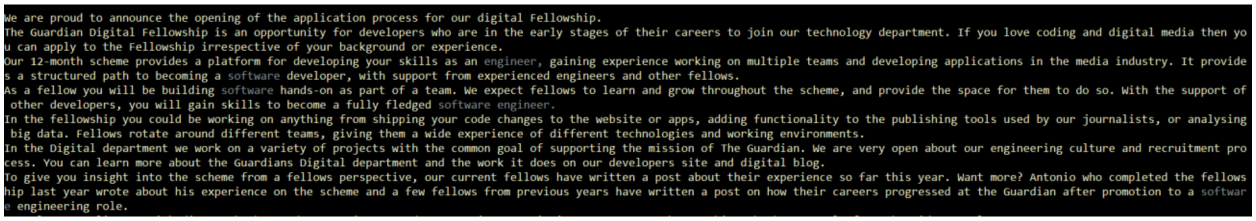


```
COMMON NAME: Giant Panda

SCIENTIFIC NAME: Ailuropoda melanoleuca

TYPE: Mammals

DIET: Omnivore

AVERAGE LIFE SPAN IN THE WILD: 20 years

SIZE: 4 to 5 feet

WEIGHT: 300 pounds
CURRENT POPULATION TREND: Increasing
ABOUT THE GIANT PANDA

The giant panda has an insatiable appetite for bamboo. A typical animal eats half the dayΓÇöa full 12 out of every 24 hoursΓÇöand relieves itself dozens of times a day. It takes 28 pounds o
f bamboo to satisfy a giant panda's daily dietary needs, and it hungrily plucks the stalks with elongated wrist bones that function rather like thumbs. Pandas will sometimes eat birds or ro
dents as well.

Behavior and Habitat

Wild pandas live only in remote, mountainous regions in central China. These high bamboo forests are cool and wetΓÇöjust as pandas like it. They may climb as high as 13,000 feet to feed on
higher slopes in the summer season.
```

b) Demo 2: AND query

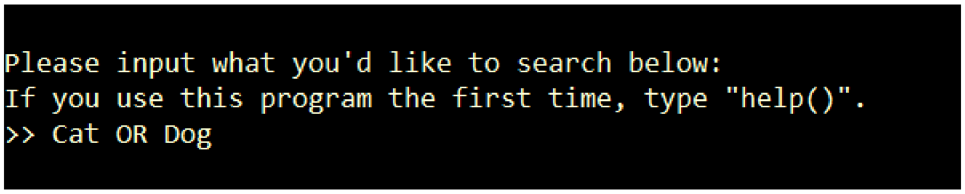


```
Please input what you'd like to search below:
If you use this program the first time, type "help()".
>> Software AND Engineer
```

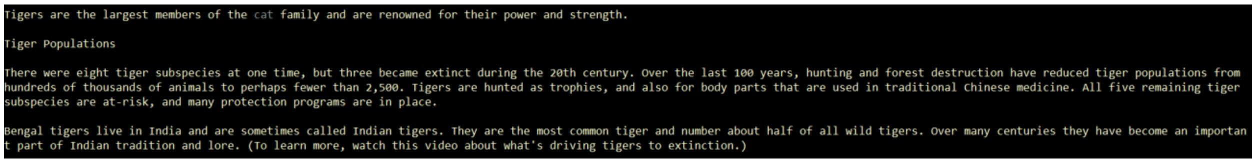


```
We are proud to announce the opening of the application process for our digital Fellowship.
The Guardian Digital Fellowship is an opportunity for developers who are in the early stages of their careers to join our technology department. If you love coding and digital media then yo
u can apply to the Fellowship irrespective of your background or experience.
Our 12-month scheme provides a platform for developing your skills as an engineer, gaining experience working on multiple teams and developing applications in the media industry. It provide
s a structured path to becoming a software developer, with support from experienced engineers and other fellows.
As a fellow you will be building software hands-on as part of a team. We expect fellows to learn and grow throughout the scheme, and provide the space for them to do so. With the support of
 other developers, you will gain skills to become a fully fledged software engineer.
In the fellowship you could be working on anything from shipping your code changes to the website or apps, adding functionality to the publishing tools used by our journalists, or analysing
 big data. Fellows rotate around different teams, giving them a wide experience of different technologies and working environments.
In the Digital department we work on a variety of projects with the common goal of supporting the mission of The Guardian. We are very open about our engineering culture and recruitment pro
cess. You can learn more about the Guardians Digital department and the work it does on our developers site and digital blog.
To give you insight into the scheme from a fellows perspective, our current fellows have written a post about their experience so far this year. Want more? Antonio who completed the fellows
hip last year wrote about his experience on the scheme and a few fellows from previous years have written a post on how their careers progressed at the Guardian after promotion to a softwar
e engineering role.
```

c) Demo 3: OR query

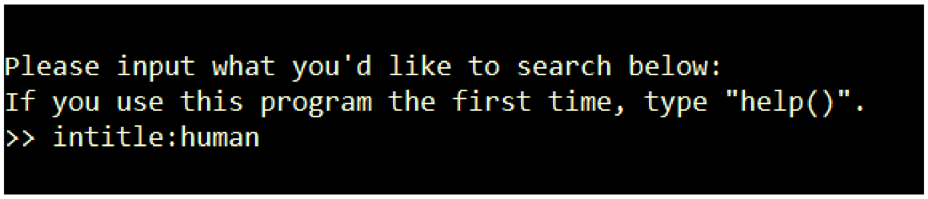


```
Please input what you'd like to search below:
If you use this program the first time, type "help()".
>> Cat OR Dog
```

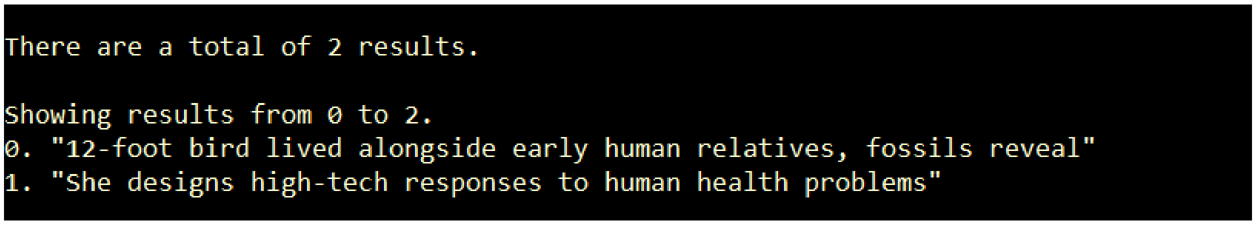


```
Tigers are the largest members of the cat family and are renowned for their power and strength.

Tiger Populations

There were eight tiger subspecies at one time, but three became extinct during the 20th century. Over the last 100 years, hunting and forest destruction have reduced tiger populations from
hundreds of thousands of animals to perhaps fewer than 2,500. Tigers are hunted as trophies, and also for body parts that are used in traditional Chinese medicine. All five remaining tiger
subspecies are at-risk, and many protection programs are in place.

Bengal tigers live in India and are sometimes called Indian tigers. They are the most common tiger and number about half of all wild tigers. Over many centuries they have become an importan
t part of Indian tradition and lore. (To learn more, watch this video about what's driving tigers to extinction.)
```

d) Demo 4: intitle query

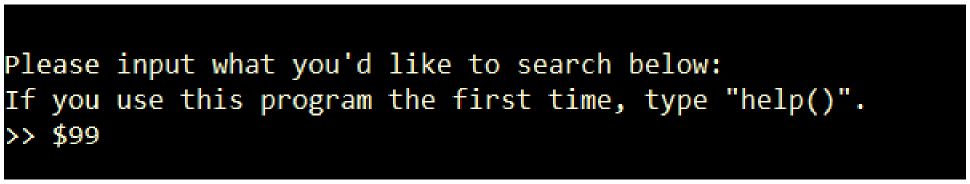


```
Please input what you'd like to search below:
If you use this program the first time, type "help()".
>> intitle:human
```

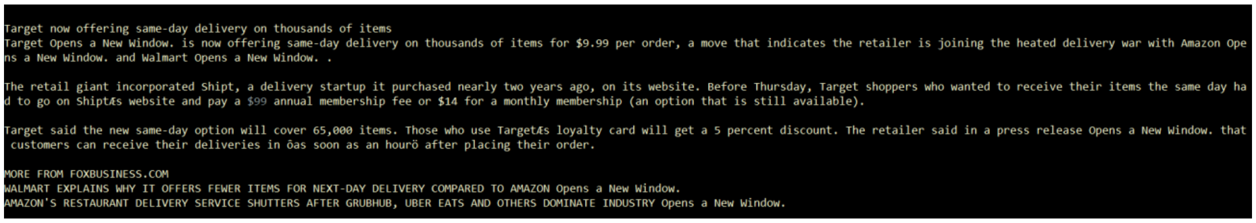


```
There are a total of 2 results.

Showing results from 0 to 2.
0. "12-foot bird lived alongside early human relatives, fossils reveal"
1. "She designs high-tech responses to human health problems"
```

e) *Demo 5: Money query*



```
Please input what you'd like to search below:
If you use this program the first time, type "help()".
>> $99
```



```
Target now offering same-day delivery on thousands of items
Target Opens a New Window. is now offering same-day delivery on thousands of items for $9.99 per order, a move that indicates the retailer is joining the heated delivery war with Amazon Ope
ns a New Window. and Walmart Opens a New Window. .

The retail giant incorporated Shipt, a delivery startup it purchased nearly two years ago, on its website. Before Thursday, Target shoppers who wanted to receive their items the same day ha
d to go on ShiptÆs website and pay a $99 annual membership fee or $14 for a monthly membership (an option that is still available).

Target said the new same-day option will cover 65,000 items. Those who use TargetÆs loyalty card will get a 5 percent discount. The retailer said in a press release Opens a New Window. that
 customers can receive their deliveries in ôas soon as an hourö after placing their order.

MORE FROM FOXBUSINESS.COM
WALMART EXPLAINS WHY IT OFFERS FEWER ITEMS FOR NEXT-DAY DELIVERY COMPARED TO AMAZON Opens a New Window.
AMAZON'S RESTAURANT DELIVERY SERVICE SHUTTERS AFTER GRUBHUB, UBER EATS AND OTHERS DOMINATE INDUSTRY Opens a New Window.
```

8) *Remark and additional features*

a) *History suggestion*

I use a history.txt file to store all the inputted queries.

Whenever the user enters a character from a keyboard, it will _getch that char and searching for up to 10 queries that start with the matching char.

To print out the history suggestion, we have to design the console, interact with the cursor to choose the place to output and back to the input line after printing the suggestion.

This process will stop when user press Enter (equivalent to character '\r') and we get an input query from the user to use in the next process.

b) *Input rule*

If you input query, you cannot type blank space at the beginning, the backspace will make no sense. You cannot also type two consecutive backspace.

c) *Help command*

You can type “help()” to get instructions how to use this program.

d) *View history command*

You can type “viewHistory()” to view all the queries you have searched.

e) *Delete history command*

You can type “clearHistory()” to delete all the queries you have searched.

f) *Exit command*

You can type “exit()” instead of enter query to exit the program.